PATENT APPLICATION

IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re the application of:	Attorney Docket No.: 4062.14US01
Jose De La Torre-Bueno	Confirmation No.: 4964
Application No.: 09/542,091	Examiner: Vikkram Bali
Filed: April 3, 2000	Group Art Unit: 2624

For: REMOTE INTERPRETATION OF MEDICAL IMAGES

DECLARATION PURSUANT TO 37 C.F.R. § 1.132

I, Paul C. Onderick, hereby declare and state as follows:

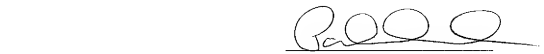
1. I am an Attorney with Patterson, Thuente, Skaar & Christensen, PA. I am a Patent Attorney registered to practice before the United States Patent and Trademark Office, Registration No. 45,354. I am over the age of 18 years and have personal knowledge of the facts that are stated herein. I am submitting this Declaration in support of traversal of a rejection for lack of enablement made in an Office Action in Application No. 09/542,091, the Office Action being dated September 27, 2007.
2. Attached hereto is exhibit 1 is a true and correct copy of a document found on the world wide web entitled "NCSA Image Map Tutorial." The document indicates that image maps are predefined coordinate ranges that can then be selected to perform other tasks. The document indicates that it is dated November 5, 1995.

3. Attached hereto is exhibit 2 is a true and correct copy of pages 131-137 of a book entitled "JavaScript, The Definitive Guide, 3rd Edition, authored by David Flanagan. According to the copyright page the book was published in June of 1998.

4. Attached hereto as exhibit 3, is a true and correct copy of pages 875-881 of a book entitled "Beginning Java2, JDK 1.3 Edition, authored by Ivor Horton. According to the copyright page the book was first published in March 2000.

I declare under penalty of perjury under the laws of the United States of America that the foregoing is true and correct. I acknowledge that willful false statements and the like are punishable by fine or imprisonment, or both (18 U.S.C. § 1001) and may jeopardize the validity of the application or any patent issuing thereon. 37 C.F.R. § 1.68.

Executed this 27th day of March, 2008.



Paul C. Onderick

EXHIBIT 1

NCSA Imagemap Tutorial

Eternal gratitude to Kevin Hughes, *kevinh@pulua.hcc.hawaii.edu*, for his code to find intersections of points with circles and polygons.

This document is a step-by-step tutorial for designing and serving graphical maps of information resources with either the external imagemap CGI script or with the built in imagemap support in NCSA HTTPd 1.5. Through such a map, users can be provided with a graphical overview of any set of information resources; by clicking on different parts of the overview image, they can transparently access any of the information resources (possibly spread out all across the Internet).

Note: Certain newer browsers support Client Side Imagemaps. For more information on Client Side Imagemaps, check out the Spyglass tutorial.

Requirements

This tutorial assumes use of NCSA HTTPd version 1.0a5 or later. To use the internal imagemap support, you need NCSA HTTPd 1.5. Some other servers (e.g. Plexus, Apache, Netsite) can also serve imagemaps, in server-specific ways; see the specific server's docs for more information.

➡ *Make sure you have a working NCSA HTTPd server installed and running.*

This tutorial also assumes use of a browser that supports inlined images, server side imagemapping and HTTP/1.0 URL Redirection.

Compiling the Imagemap Script

If you downloaded or pre-compiled NCSA HTTPd server, or compiled your own, chances are the imagemap program was automatically compiled and added to your cgi-bin directory. If so, you can skip this step. If not, you can download the current imagemap.c from here.

Compile the imagemap script by first changing into your ServerRoot directory, and then into the cgi-src subdirectory. Put the new imagemap.c source in place of the old one (if you have the old version of imagemap.c) and then, type `make imagemap` and you should be all set.

Your First Imagemap

In this section we walk through the steps needed to get an initial image map up and running.

➡ *First: create an image.*

There are a number of image creation and editing programs that will work nicely -- the one I use is called *xpaint* (I don't know where its home page is anymore). You can get it here. It's for UNIX systems running an X interface.

➡ *Second: create an imagemap map file.*

This file maps regions to URLs for the given image. For a list of tools that may help you create a map file, see Yahoo's Imagemap Directory. For instance, there is a program called mapedit that you could use.

A common scheme for an imagemap is a collection of points, polygons, rectangles and circles, each containing a short text description of some piece of information or some information server; interconnections are conveyed through lines or arcs. Try to keep the individual items in the map spaced out far enough so a user will clearly know what he or she is clicking on.

Lines beginning with # are comments. Every other non-blank line consists of the following:

```
method URL coord1 coord2 ... coordn
```

method is one of the following:

- **default** - For the default URL
  - Coordinates: none
- **circle** - For a circle
  - Coordinates: `center edgepoint`
- **poly** - For a polygon of at most 100 vertices
  - Coordinates: Each coordinate is a vertex.
- **rect** - For a rectangle
  - Coordinates: `upper-left lower-right`
- **point** - For closest to a point
  - Coordinate: `thePoint`

URL is one of the following:

- A Local URL
  - ex. `/docs/tutorials`
- A Full URL
  - ex. `http://www.yahoo.com/`

coord#

Each coord entry is a coordinate, format x,y. The number depends on the **method**.

Notes:

- each method is evaluated in the order it is placed in the configuration file. If you have overlapping areas, such as a circle inside of a rectangle, you should place whichever one you want evaluated first before the other in the map file. In this case, we would put the circle before the rectangle.

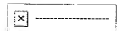
- Also note that it does not make sense to use the default method with the point method because if even one point method is specified, anywhere you click will be considered close to the point and the URL specified by point will be serviced.

- If you will be serving access authentication-protected documents through your imagemap, you MUST use fully qualified URLs, for example

  ```
  http://your.server.com/path/to/protected/file.html
  ```

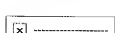
  otherwise access will be denied.

Here is an example imagemap linked from the Mosaic Demo page. Here is what a map file looks like:



```
default /X11/mosaic/public/none.html

rect http://cui_www.unige.ch/w3catalog    15,8    135,39
rect gopher://rs5.loc.gov/11/global       245,86  504,143
rect http://nearnet.gnn.com/GNN-ORA.html  117,122 175,158
```



The format is fairly straightforward. The first line specifies the default response (the file to be returned if the region of the image in which the user clicks doesn't correspond to anything).

Subsequent lines specify rectangles in the image that correspond to arbitrary URLs -- for the first of these lines, the rectangle specified by `15,8` (`x,y` of the upper-left corner, in pixels) and `135,39` (lower-right corner) corresponds to URL `http://cui_www.unige.ch/w3catalog`.

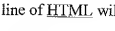
So, what you need to do is find the upper-left and lower-right corners of a rectangle for each information resource in your image map. A good tool to use on UNIX for this is *xv* (also on **ftp.x.org** in **/contrib**) -- pop up the Info window and draw rectangles over the image with the middle mouse button.

It doesn't matter what you name your map file, **but it does matter where you put it!** Specifically, you cannot have your mapfile reside in the in the top-level of the DocumentRoot, because the imagemap CGI program will not see a "/" in the extra `PATH_INFO` that you are referencing. If this doesn't make sense to you, then just trust me: place your map file in a subdirectory.

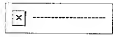
➡ *Third: referencing your imagemap in your HTML file.*

To reference your new map, you construct URLs pointing to it.

For example, if your username is `newbie` and you have created a `sample.map` file in the directory called `foo` in your `public_html` home directory, and used the image `sample.gif` for the map, the following line of HTML will reference it:



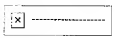
```
<A HREF="http://www.school.edu/cgi-bin/imagemap/~newbie/foo/sample.map">
<IMG SRC="/~newbie/gifs/sample.gif" ISMAP>
</A>
```



The extra `PATH_INFO` after the `/cgi-bin/imagemap` tells the imagemap program where to find your map file. You'll note that it is possible to request the map file itself by simply requesting:

```
http://www.school.edu/~newbie/foo/sample.map
```

Or, using the internal imagemap support of NCSA HTTPd 1.5:



```
<A HREF="http://www.school.edu/~newbie/foo/sample.map">
<IMG SRC="/~newbie/gifs/sample.gif" ISMAP>
</A>
```

➡ *Fourth: Try it out!*

Load the HTML file, look at the inlined image, click somewhere, and see what happens.

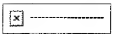
Internal Imagemap Support

With NCSA HTTPd 1.5, the server can now handle imagemap queries without launching the external imagemap script. This provides for a faster mechanism and decreased server load.

In order to enable the internal imagemap code, the server must be compiled with `IMAGEMAP_SUPPORT` defined. This is the default. Also, you need to add the magic mime type for the imagemap files in much the same way you would for CGI scripts:

```
AddType text/x-imagemap .map
```

Then, you can reference the map file directly in your HTML instead of through the imagemap script, ie:



```
<A HREF="http://www.school.edu/~newbie/foo/sample.map">
<IMG SRC="/~newbie/gifs/sample.gif" ISMAP>
</A>
```

A Complete Example

The fish demo in another section of this manual used the following configuration files:

Map Configuration File

The map configuration file used for this picture was rather lengthy. I used xv to get the coordinates.

Reference the map in HTML document:

```
<A HREF="/cgi-bin/imagemap/docs/info/fish.map"><IMG SRC="fish33.gif" ISMAP> <A>
```

Real-World Examples

Following are examples of distributed imagemaps on servers in the real world; they may or may not work at any point in time.

- Experimental Internet Resources Metamap at NCSA.
- University of California Museum of Paleontology.
- National Institute of Standards and Technology.
- Server map at NCHPC Information Server.

For More Information

For more information, see the NCSA HTTPd documentation.

 Return to the tutorial index

Return to administration overview

NCSA HTTPd Development Team / httpd@ncsa.uiuc.edu / 11-5-95

EXHIBIT 2

JavaScript: The Definitive Guide, Third Edition
by David Flanagan

Published by O'Reilly & Associates, Inc., 101 Morris Street, Sebastopol, CA 95472.

Editor: Paula Ferguson

Production Editor: Madeleine Newell

Printing History:

August 1996:	Beta Edition.
January 1997:	Second Edition.
June 1998:	Third Edition.

This book is printed on acid-free paper with 85% recycled content, 15% post-consumer waste. O'Reilly & Associates is committed to using paper with the highest recycled content available consistent with high quality.

ISBN: 1-56592-392-8 [5/99]

Methods have one very important property: the object a method is invoked through becomes the value of the `this` keyword within the body of the method. For example, when we invoke `o.m()`, the body of the method can refer to the object `o` with the `this` keyword.

The discussion of the `this` keyword should begin to make it clear why we use methods at all. Any function that is used as a method is effectively passed an extra argument—the object through which it is invoked. Typically, a method performs some sort of operation on that object, so the method invocation syntax is a particularly elegant way to express the fact that a function is operating on an object. Compare the following two lines of code:

```
rect.setSize(x, y);
setRectSize(rect, x, y);
```

The two lines may perform exactly the same operation on the object `rect`, but the method invocation syntax shown first more clearly indicates the idea that it is the object `rect` that is the primary focus, or target, of the operation. (If the first line does not seem a more natural syntax to you, you are probably new to object-oriented programming. With a little experience, you will learn to love it!)

While it is useful to think of functions and methods differently, there is not actually as much difference between them as there initially appears to be. Recall that functions are values stored in variables and that variables are nothing more than properties of a global object. Thus, when you invoke a function, you are actually invoking a method of the global object. Within such a function, the `this` keyword refers to the global object. Thus, there is no technical difference between functions and methods. The real difference lies in design and intent: methods are written to operate somehow on the `this` object, while functions usually stand alone and do not use the `this` object.

The typical usage of methods is more clearly illustrated through an example. Example 8-2 returns to the Rectangle objects of Example 8-1 and shows how a method that operates on Rectangle objects can be defined and invoked.

Example 8-2: Defining and Invoking a Method

```
// This function uses the this keyword, so it doesn't make sense to
// invoke it by itself; it needs instead to be made a method of some
// object that has "width" and "height" properties defined.
function compute_area()
{
    return this.width * this.height;
}

// Create a new Rectangle object, using the constructor defined earlier.
var page = new Rectangle(8.5, 11);
```

Example 8-2: Defining and Invoking a Method (continued)

```
// Define a method by assigning the function to a property of the object.
page.area = compute_area;

// Invoke the new method like this:
var a = page.area();     // a = 8.5*11 = 93.5
```

There is a shortcoming that is evident in Example 8-2: before you can invoke the `area()` method for the `rect` object, you must assign that method to a property of the object. While we can invoke the `area()` method on the particular object named `page`, we can't invoke it on any other Rectangle objects without first assigning the method to them. This quickly becomes tedious. Example 8-3 defines some additional Rectangle methods and shows how they can automatically be assigned to all Rectangle objects with a constructor function.

Example 8-3: Defining Methods in a Constructor

```
// First, define some functions that will be used as methods.
function Rectangle_area() { return this.width * this.height; }
function Rectangle_perimeter() { return 2*this.width + 2*this.height; }
function Rectangle_set_size(w,h) { this.width = w; this.height = h; }
function Rectangle_enlarge() { this.width *= 2; this.height *= 2; }
function Rectangle_shrink() { this.width /= 2; this.height /= 2; }

// Then define a constructor method for our Rectangle objects.
// The constructor initializes properties and also assigns methods.
function Rectangle(w, h)
{
    // Initialize object properties.
    this.width = w;
    this.height = h;

    // Define methods for the object.
    this.area = Rectangle_area;
    this.perimeter = Rectangle_perimeter;
    this.set_size = Rectangle_set_size;
    this.enlarge = Rectangle_enlarge;
    this.shrink = Rectangle_shrink;
}
// Now, when we create a rectangle, we can immediately invoke methods on it:
var r = new Rectangle(2,2);
var a = r.area();
r.enlarge();
var p = r.perimeter();
```

The technique shown in Example 8-3 also has a shortcoming. In this example, the `Rectangle()` constructor sets seven properties of each and every Rectangle object it initializes, even though five of those properties have constant values that are the same for every rectangle. Each property takes up memory space; by adding methods to our Rectangle class, we've more than tripled the memory requirements of each Rectangle object. Fortunately, JavaScript 1.1 introduced a solution to this problem: it allowed an object to inherit properties from a prototype object. The next section describes this technique in detail.

8.4 Prototypes and Inheritance

We've seen how inefficient it can be to use a constructor to assign methods to the objects it initializes. When we do this, each and every object in the class has identical copies of the same method properties. In JavaScript 1.1 and later, there is a much more efficient way to specify methods, constants, and other properties that are shared by all objects in a class.

JavaScript 1.1 introduced the notion of a *prototype object*. Every object has a prototype; an object inherits all of the properties of its prototype. This means that all of the properties of the prototype object appear to be properties of the objects that inherit them. To specify the prototype object for a class of objects, we set the value of the `prototype` property of the constructor function to the appropriate object. Then, when a new object is initialized with the constructor, JavaScript automatically uses the specified object as the prototype for the newly created object.

A constructor defines a class of objects and initializes properties, such as `width` and `height`, that are the state variables for the class. The prototype object is associated with the constructor, so each member of the class inherits exactly the same set of properties from the prototype. This means that the prototype object is an ideal place for methods and other constant properties.

Note that inheritance occurs automatically as part of the process of looking up a property value. Properties are *not* copied from the prototype object into new objects; they merely appear as if they were properties of those objects. This has two important implications. First, the use of prototype objects can dramatically decrease the amount of memory required by each object, since it can inherit many of its properties. The second implication is that an object inherits properties even if they are added to its prototype *after* the object is created.

Each class has one prototype object, with one set of properties. But there are potentially many instances of a class, each of which inherits those prototype properties. Because one prototype property can be inherited by many objects,

JavaScript must enforce a fundamental asymmetry between reading and writing property values. When you read property p of an object o, JavaScript first checks to see if o has a property named p. If it does not, it next checks to see if the prototype object of o has a property named p. This is what makes prototype-based inheritance work.

When you write the value of a property, on the other hand, JavaScript does not use the prototype object. To see why, consider what would happen if it did: suppose you try to set the value of the property `o.p` when the object o does not have a property named p. Further suppose that JavaScript goes ahead and looks up the property p in the prototype object of o and allows you to set the property of the prototype. Now you have changed the value of p for a whole class of objects—not at all what you intended.

Therefore, property inheritance occurs only when you read property values, not when you write them. If you set the property p in an object o that inherits that property from its prototype, what happens is that you create a new property p directly in o. Now that o has its own property named p, it no longer inherits the value of p from its prototype. When you read the value of p, JavaScript first looks at the properties of o. Since it finds p defined in o, it doesn't need to search the prototype object and never finds the value of p defined there. We sometimes say that the property p in o "shadows" or "hides" the property p in the prototype object. Prototype inheritance can be a confusing topic. Figure 8-1 illustrates the concepts we've discussed here.

Because prototype properties are shared by all objects of a class, it generally makes sense to use them only to define properties that are the same for all objects within the class. This makes prototypes ideal for defining methods. Other properties with constant values (such as mathematical constants) are also suitable for definition with prototype properties. If your class defines a property with a very commonly used default value, you might define this property and its default value in a prototype object. Then, the few objects that want to deviate from the default value can create their own private, unshared copies of the property and define their own non-default values.

Let's move from an abstract discussion of prototype inheritance to a concrete example. Suppose we define a `Circle()` constructor function to create objects that represent circles. The prototype object for this class is `Circle.prototype`, so we can define a constant available to all Circle objects like this:

```
Circle.prototype.pi = 3.14159;
```

The prototype object of a constructor is created automatically by JavaScript. In most versions of JavaScript, every function is automatically given an empty prototype object, just in case it is used as a constructor. In Navigator 3, however, the

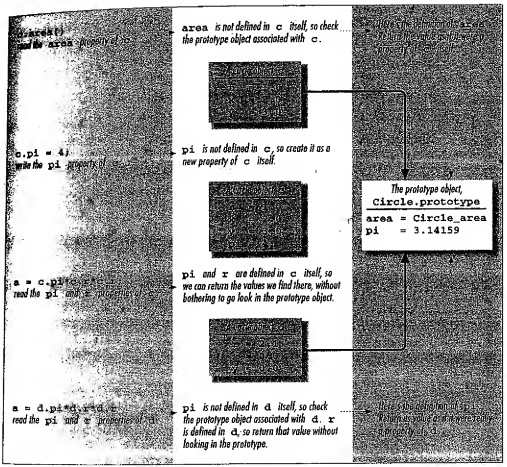


Figure 8-1: Objects and prototypes

prototype object is not created until the function is used as a constructor for the first time. What this means is that for compatibility with Navigator 3, you should create at least one object of a class before you use the prototype object to assign methods and constants to objects of that class. So, if we have defined a `Circle()` constructor, but not yet used it to create any Circle objects, we'd define the constant property `pi` like this:

```
// First create and discard a dummy Circle object.
// This forces Navigator 3 to create the prototype object.
new Circle();

// Now we can set properties in the prototype.
Circle.prototype.pi = 3.14159;
```

Example 8-4 shows our Circle example fully fleshed out. The code defines a `Circle` class by first defining a `Circle()` constructor to initialize each individual

object, and then by setting properties of `Circle.prototype` to define methods and constants shared by all instances of the class.

Example 8-4: Defining a Circle Class with a Prototype Object

```
// Define a constructor method for our class.
// Use it to initialize properties that will be different for
// each individual circle object.
function Circle(x, y, r)
{
    this.x = x;  // The X-coordinate of the center of the  circle
    this.y = y;  // The Y-coordinate of the center of the circle
    this.r = r;  // The radius of the circle
}

// Create and discard an initial Circle object.
// This forces the prototype object to be created in Navigator 3.
new Circle(0,0,0);

// Define a constant: a property that will be shared by
// all circle objects. Actually, we could just use Math.PI,
// but we do it this way for the sake of example.
Circle.prototype.pi = 3.14159;

// Define a method to compute the circumference of the circle.
// First declare a function, then assign it to a prototype property.
// Note the use of the constant defined above.
function Circle_circumference() { return 2 * this.pi * this.r; }
Circle.prototype.circumference = Circle_circumference;

// Define another method. This time we use the Function()
// constructor to define the function and assign it to a prototype
// property all in one step.
Circle.prototype.area = new Function("return this.pi * this.r * this.r;");

// The Circle class is defined.
// Now we can create an instance and invoke its methods.
var c = new Circle(0.0, 0.0, 1.0);
var a = c.area();
var p = c.circumference();
```

8.4.1 Prototypes and Built-In Classes

It is not only user-defined classes that have prototype objects. Built-in classes, such as String and Date, have prototype objects too, and you can assign values to them. (This does not work in IE 3, however).

For example, the following code defines a new method that is available for all String objects:

```
// Returns true if the last character is c
String.prototype.endsWith = function(c) {
    return (c == this.charAt(this.length-1))
}
```

Having defined the new `endsWith()` method in the String prototype object, we can use it like this:

```
var message = "hello world";
message.endsWith('h')       // Returns false
message.endsWith('d')       // Returns true
```

8.5 Object-Oriented JavaScript

Although JavaScript supports a data type we call an object, it does not have a formal notion of a class. This makes it quite different from classic object-oriented languages such as C++ and Java. The common conception about object-oriented programming languages is that they are strongly typed and support class-based inheritance. By these criteria, it is easy to dismiss JavaScript as not being a true object-oriented language. On the other hand, we've seen that JavaScript certainly makes heavy use of objects, and we've seen that it has its own type of prototype-based inheritance. The truth is that JavaScript is a true object-oriented language. It draws inspiration from a number of other (relatively obscure) object-oriented languages that use prototype-based inheritance instead of class-based inheritance.

Although JavaScript is not a class-based object-oriented language, it does a good job of simulating the features of class-based languages like Java and C++. I've been using the term class informally throughout this chapter. This section more formally explores the parallels between JavaScript and true class-based inheritance languages like Java and C++.

Let's start by defining some basic terminology. An *object*, as we've already seen, is a data structure that contains various pieces of named data and may also contain various methods to operate on those pieces of data. An object groups related data values and methods into a single convenient package, which generally makes programming easier by increasing the modularity and reusability of code. Objects in JavaScript may have any number of properties, and properties may be added to an object dynamically. This is not the case in strictly typed languages like Java and C++. In those languages, each object has a predefined set of properties (or fields, as they are often called), where each property is of a predefined type. When we

EXHIBIT 3

Beginning Java 2 – JDK 1.3 Edition

Printing History

First Published March 2000

Published by Wrox Press Ltd. Arden House, 1102 Warwick Road, Acocks Green,
Birmingham, B27 6BH, UK
Printed in Canada

ISBN 1-861003-66-8

Outside the loop we initialize a variable `t` that stores the time interval in seconds as a value of type `float`. We will use this in our calculations for the image position and velocity. We also define a constant `g`, and initialize the acceleration, `a`, and the velocity `v`, of the image. Within the loop we just apply the calculations that we discussed earlier.

The image is compressed when the y coordinate, `imageY`, is positive. This is because the top of the applet is where y is 0, and the bottom of the applet is where y is `imageHeight`, so when the top-left corner of the image is below the top of the applet, it is being squashed. In this case, if the image is still heading downwards, we apply an acceleration in the opposite direction that is the value of the expression `-10.000f*g*imageY/imageHeight`. This expression is arbitrary, but it has the effect of slowing the image down more and more as it is compressed. When `imageY` is negative, the image is off the ground so we set the acceleration back to `g`.

Displaying the Image

The image is displayed by the `ImagePanel` object, so we need to implement the `paint()` method for this class to display the image normally when `imageY` is zero or negative, and deal with squashing the image when `imageY` is positive. This is going to be easy since we can use the `drawImage()` method we used in the previous example to draw the image normally, and use an overloaded version that is specifically intended for scaling an image on the fly to fit a particular area. The overloaded method has the following arguments:

g,

When you call this method, the image is scaled on the fly to fit the destination area specified by the arguments. Since you specify the coordinates of the top-left of the image, as well as its width and height, it is possible to use this method to display part of an image, and fit it to the destination space that you specify. Like all the `drawImage()` methods, this method can draw part of an image when loading of the image is not complete. In this case the method returns `false`. The `ImageObserver` that is passed as the last argument – usually the applet itself – is notified when more of the image becomes available, with the result that the image is repainted. When the entire image has been drawn, the `drawImage()` method returns `true`.

It would also be useful to color the background with a color that contrasts with the image. With this in mind we can implement the `paint()` method for the `ImagePanel` class as:

```
public void paint(Graphics g)
{
```

```
        g2d.setPaint(Color.lightGray);                    // Set a background
        g2d.fillRect(0, 0, areaWidth, imageHeight);       // Paint background
        if(imageY > 0)
          g2d.drawImage(image, imageX, imageY, this);     // Draw image
        else
          g2d.drawImage(image,
                        imageX, imageY, imageWidth, imageHeight,
                        0, 0, imageWidth, imageHeight,
                        this);
      }
```

The `fillRect()` method fills the entire area of the applet with the color that we set in the `setPaint()` call, `Color.lightGray`. When `imageY` is greater than 0, we use the version of `drawImage()` that scales the image on the fly to fit the area available, from the `imageY` position to the bottom of the applet at the y coordinate, `imageHeight`.

It is worth noting that a `JPanel` object is double-buffered by default. This means that all rendering is to a new image that is to be displayed is done in a buffer in memory, and only when image is complete are the pixels for the entire picture written to the screen. Since the existing image that is displayed is not altered while the new image is being created, this eliminates the flicker and flashing that can occur if your display buffer is updated incrementally while it is displayed.

If you have put together all the bits of code we have discussed, you should have a working applet.

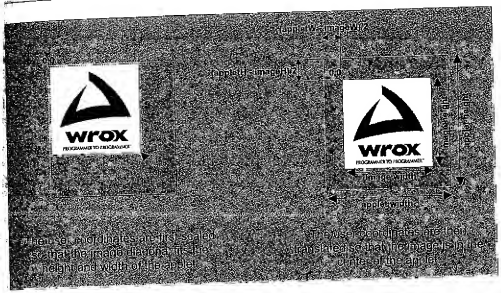
Transforming Images

We have already seen back in Chapter 15 that we can apply a transformation to a graphics context to modify the user coordinate system relative to the device coordinate system. The transformation can be a translation, a rotation, a scaling operation, a shearing operation or a combination of all four. Of course such a transformation applies to images that you draw as well as anything else.

In our previous example we adjusted the size of the applet to accommodate the image. In many situations you would not want to do this. Typically, there are likely to be all kinds of things on the page so you would want your applet to keep within the space allotted to it. We could have done this by scaling the image to fit the space available. Rather than go over the old ground let's create a new applet to try this out, and to add a bit of spice this time we will spin the image about its center, rather than dropping it. That way we will get to use a more complicated transform.

Try It Out – Spinning an Image

This applet will create an animation that spins the image about its center point. We will therefore need to make the diagonal of the image fit within both the height and width of the applet if we want to see all of it as it rotates. We also want the scaled image to fit in the center of the applet, so we will translate the user space after we have applied the scale transform.



e the image is loaded, we can calculate the length of the diagonal of the image as the square root of
sum of the squares of the width and height. We can then calculate the scale factor that we require by
ding the width and height of the applet by the diagonal of the image, and taking the minimum of
e two values. We can create an `AffineTransform` object in the `init()` method that combines
th the scaling and the translation, and then just apply it in the `paint()` method for the `ImagePanel`
lass.

his applet class will contain the same methods as the previous example but with different
mplementations. The rotation will be accomplished by an additional transformation. We will also have
n inner `ImagePanel` class to define the panel that will display the image. This will only differ in the
mplementation of the `paint()` method.

et's start with the `init()` method and the data members in the `Applet` class. The loading of the
image will be identical to the previous example, so we need the `image` and `tracker` members too.
After the image has been loaded, we will calculate the scaling and translation that is necessary.

Here's the applet class with its data members and the `init()` method:

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;
import java.awt.geom.*;
import javax.swing.*;

public class ImageScaling extends JApplet
{
  // This method is called when the applet is loaded
  public void init()
  {
    tracker = new MediaTracker(this);
    Image image = null;
```

```
      try
      {
        // Image from a file specified by a URL
        image = getImage(new URL(getCodeBase(),"Images/wrox_logo.gif"));
      }
      catch(MalformedURLException e)
      {
        System.out.println("Failed to create URL:\n" + e);
      }
      tracker.addImage(image,0);                        // Load image
      try
      {
        tracker.waitForAll();                           // Wait for image to load
        if(tracker.isErrorAny())                        // If there is an error
          return;                                       // give up

        Dimension size = getSize();                     // Get applet
        imageWidth = image.getWidth(this);              // and image
        imageHeight = image.getHeight(this);            // dimensions

        // Calculate scale factor as diagonal of image into width and height
        double diagonal = Math.sqrt(
                          imageWidth*imageWidth + imageHeight*imageHeight);
        double scaleFactor = Math.min(size.width/diagonal, size.height/diagonal);

        // Create a transform to translate and scale the image
        AffineTransform translation = AffineTransform.getTranslateInstance(
                                      (size.width-imageWidth*scaleFactor)/2,
                                      (size.height-imageHeight*scaleFactor)/2);
        translation.scale(scaleFactor, scaleFactor);

        imagePanel = new ImagePanel(image);             // Create panel showing the image
        getContentPane().add(imagePanel);               // Add the panel to the content pane
      }
      catch(InterruptedException e)
      {
        System.out.println(e);
      }
    }

    MediaTracker tracker;                               // Tracks image loading
    ImagePanel imagePanel;
    int imageWidth, imageHeight;                        // Image dimensions
  }
```

The unshaded code is exactly the same as in the previous method. The `AffineTransform` member, `at`, stores a transform that scales and translates the image. The `angle` member will store the rotation angle in radians that will be calculated in the `run()` method for the `whirler` thread, and applied in the `paint()` method for the `imagePanel` object. We have made this is a member of the class rather than declare it as a local variable in the `run()` method so we can pick up the value when the applet is stopped and restarted. The other fields that follow `angle` are all concerned with orienting and drawing the image.

The constant, `INTERVAL`, stores the time interval between one instance of drawing the image and the next. We store the time for a complete rotation of the image through 360 degrees, which is 2π radians, in `ROTATIONTIME`. The variable `STEPS_PER_ROTATION` holds the number of steps for a complete rotation, so with the values we have set for the previous two variables, this will be 40. Finally, the variable, `stepCount`, will accumulate the total number of steps modulo `STEPS_PER_ROTATION`. The methods to start and stop the animation thread are the same as in the previous example, apart from the new names for the thread and the control variable:

```
  // This method is called when the browser starts the applet
  public void start()
  {
    if(tracker.isErrorAny())                      // If any image errors
      return;                                     // don't create the thread
    whirler = new Thread(this);                   // Create the animation thread
    whirling = true;
    whirler.start();                              // and start it
  }

  // This method is called when the browser want to stop the applet
  // when is it not visible for example
  public void stop()
  {                                               // Stop the animation loop
    whirling = false;                             // Discard the thread
    whirler = null;
  }
```

The thread code itself will be very similar to the previous example – the timing mechanism is exactly the same. We now need to increment the rotation angle in each time interval so it will be much simpler:

```
  // This method is called when the animation thread is started
  public void run()
  {
    long time = System.currentTimeMillis();              // Starting time

    // Move image while whirling is true
    while(whirling)
    {
      imagePanel.repaint();                       // Repaint the image

      // Wait until the end of the interval
      try
      {
        time += INTERVAL;                         // Increment the time
        angle = 2.0*Math.PI*stepCount++/ STEPS_PER_ROTATION;
        stepCount %= STEPS_PER_ROTATION;
        Thread.sleep(Math.max(0, time - System.currentTimeMillis()));
      }
```

```
            catch (InterruptedException e)
            {
             break;
            }
          }
        }
```

The `stepCount` variable starts at 0 and is incremented by 1 on each iteration of the loop. Since a complete rotation through 2 radians should occur after `STEPS_PER_ROTATION` steps, after `step` steps the rotation angle is the result of the expression `2.0*Math.PI*stepCount/STEPS_PER_ROTATION`. In the statement calculating the rotation we also increment `stepCount` using the postfix increment operator. The image returns to its original position after `STEPS_PER_ROTATION` steps, so we maintain the value of `stepCount` modulo `STEPS_PER_ROTATION`.

The last bit we need to complete the applet is the `ImagePanel` class, and this only differs from the previous example in the implementation of the `paint()` method:

```
class ImagePanel extends JPanel
{
  public ImagePanel(Image image)
  {
    this.image = image;
  }

  public void paint(Graphics g)
  {
    Graphics2D g2D = (Graphics2D)g;
    g2D.translate(50,50);

    g2D.setPaint(Color.lightGray);
    g2D.fillRect(0, 0, imageWidth, imageHeight);

    g2D.rotate(angle, imageWidth/2, imageHeight/2);

    g2D.drawImage(image, 0, 0, this);
  }

  Image image;                                    // The image
}
```

That's the complete applet so give it a whirl. It would be a good idea to make the applet larger in the html file – 200x200 say – then you can see the image more clearly. The larger applet is that it will take more processor time since there are more pixels to

How It Works

The basic principles are the same as in the previous example. The thread code in the repaints the `imagePanel` object every `interval` milliseconds. The `while` loop increments `angle` each time, and `angle` defines the rotation transformation that `paint()` method for the `imagePanel` object.

...e the rotation with the statement:

```
...rotate(angle, imageWidth/2.0, imageHeight/2.0);      // Rotate about center
```

...nsformation specified by this version of the `rotate()` method is concatenated with the existing ...m for the graphics context, which is the `AffineTransform` object that we create in the ... method for the applet. The transform created by this `rotate()` call is a composite of a ...tion to the center of the image, the coordinates of which are defined by the last two arguments, a ...n through `angle` radians – supplied as the first argument, then a translation back to the original ...point. Thus the rotation is about the center of the image.

Using Timers

...ave adopted a do-it-yourself approach to timing when we need to redraw in animation. This ...vides a good insight into how animations operate but we can accomplish the same result rather more ...ply by making use of an object of the `Timer` class and objects of its associated `TimerTask` class. ...e classes are defined in the `java.util` package. The `Timer` class we are discussing here schedules ...operation that you define with a `TimerTask` object, either once after a given delay, or repeatedly ...h a given time interval between successive executions of the task. The task that is executed by the ...erTask` object runs in a separate thread.

...aware that there is another `Timer` class defined in `javax.swing` that provides a capability that ...pears somewhat similar at first sight but that has significant differences in the way that it works. The ...er class defined in the `javax.swing` package notifies its listeners (of type `ActionListener`) ...en a given time interval has passed, and it is up to the listener objects to carry out or initiate the task ...be executed. As with the other `Timer` class, you can use a `Timer` object do something just once after ...given interval, or repeatedly after successive intervals of time. A major difference is that the listener ...bject methods will execute on the same thread unless you provide code to ensure that is not the case.

The `Timer` and `TimerTask` combination of classes provide a way of executing repeated tasks that is ...asy to apply to animations so we will concentrate on those. While we will be applying them to ...imations here, keep in mind that you can use these methods for executing any kind of task repeatedly ...or after a fixed delay. We will start by looking at how you use a `Timer` object to schedule a task.

Timer Objects

You can use a single `Timer` object to schedule several different tasks, where each task will be defined by its own `TimerTask` object. Each `TimerTask` object defines a separate thread, so when you schedule multiple tasks they will each execute in a separate thread. The `Timer` class has been designed to allow large numbers of tasks to be executed concurrently – thousands, according to the documentation – without creating undue task scheduling overhead.

There are two constructors for `Timer` objects. The default constructor simply defines a `Timer` object that has an associated thread that is not run as a daemon thread. You will recall that a daemon thread is subordinate to the thread that created it and dies when its creator dies, whereas a non-daemon thread runs completely independently. You can make the `Timer` thread daemon by creating the `Timer` object using the constructor that accepts an argument of type `boolean`, and specifying the argument as `true`. For instance, the following statement creates a `Timer` object with a daemon thread:

```
Timer clock = new Timer(true);  // Create a daemon Timer
```